Programming Languages
– Trends in Education

David A Watt

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: daw@dcs.gla.ac.uk.

Abstract

Programming languages play a central role in the computer science curriculum. Programming is of course a fundamental component of the curriculum, and the choice of languages in which students learn to program has a major impact, both on the way that they learn and on the programming skills that they will take into industry. The past has seen considerable tension between the educational sector, in which we prefer to teach clean 'structured' languages, and the industrial sector, whose language preferences are more pragmatic. Today we see a convergence: object-oriented methods in general have emerged as the technology of choice in both sectors, and JAVA [1] in particular is a fine language to learn programming and a useful preparation for programming in other languages such as C or C++.

A different but related issue is education *about* programming languages. This is often seen as a relatively specialized topic in the computer science curriculum. However, I believe that all software engineers derive both intellectual and practical benefits from this part of their education. Few software engineers ever write compilers and interpreters, but all use them routinely, and all benefit from an understanding of how their programming constructs are actually implemented. Even fewer software engineers ever design a language, but all will learn new languages throughout their careers, and an understanding of language concepts gives them an intellectual framework that facilitates efficient learning of new languages.

1 Introduction

Programming languages play a central role in every computer science (CS) curriculum. Students are taught one or more programming languages early in the curriculum, as a means to the end of learning how to program. Later, depending on the curriculum, students may have an opportunity to study programming languages as artifacts of interest in their own right.

[1] JAVA is a trademark of Sun Microsystems Inc.

In my view, programming and the study of programming languages strongly reinforce one another. A mastery of programming deepens the student's understanding of the particular programming languages that he or she is using. Conversely, an understanding of the concepts, implementation, and specification of programming languages improves the student's mastery of programming. In this paper I shall develop these arguments in detail. I shall pay particular attention to past and current trends in education (and in industry): although the discipline has matured and its fundamental principles are unchanging, interest in particular programming languages, paradigms, and implementation methods do vary over time.

The paper is structured as follows. Section 2 is an overview of those parts of the CS curriculum in which programming languages are most prominent. Section 3 discusses programming courses, focusing on the choice of language(s) for learning how to program. The following three sections discuss courses in which programming languages are studied in their own right: Section 4 discusses concepts, Section 5 implementation, and Section 6 specification. Section 7 concludes.

2 Programming Language Topics in the Computer Science Curriculum

CS curricula are of course many and varied. The discipline has grown so fast that no curriculum can cover every possible topic. Nevertheless, there are certain topics that most CS educators would regard as essential, or at least highly desirable.

In this paper I will confine my attention to those parts of a CS curriculum in which programming languages are most prominent. Figure 1 summarizes these topics, together with their curricular relationships.

For the sake of simplicity, I will write as if each topic is covered by a complete course or module. However, a one-to-one relationship between topics and courses is not essential. Depending on how much space is allotted to each topic within a particular curriculum, a topic may be covered by more than one course, or multiple topics may be covered by a single course.

Programming 1 is an introductory course on programming, in some suitable language. This is typically the first course in the CS curriculum. *Programming 2* follows in the sequence, typically introducing wider topics such as algorithms and data structures. *Programming 2* might introduce a new programming language, if this is considered necessary or desirable.

The programming sequence should lead on to a more advanced course, *Software Engineering*, covering requirements, analysis, design, coding, test-

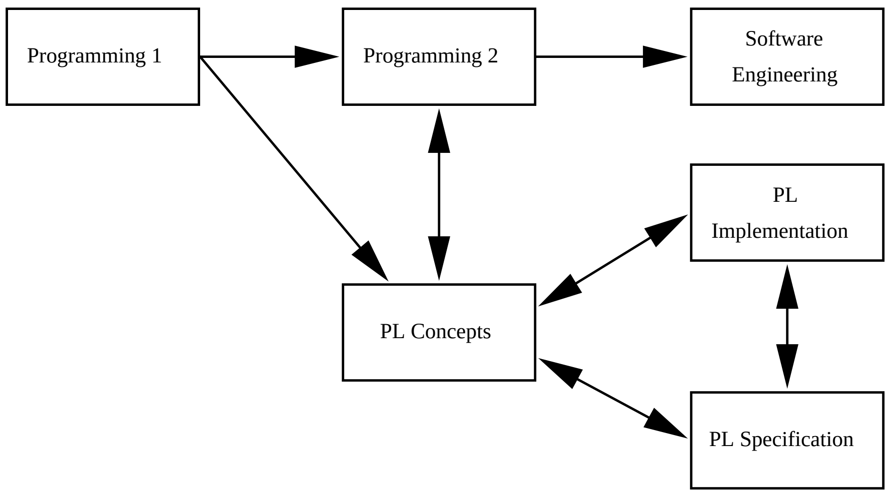

Fig. 1. Programming language (PL) topics in the CS curriculum.

ing, and maintenance.[2] *Software Engineering* lies outside the scope of this paper, but it is important that students should learn that programming is only one phase (and not even the costliest phase) of software engineering; and it equally important that educators design programming courses, and courses about programming languages, with this point in mind.

Programming Language Concepts should expose students to the fundamental concepts and paradigms in terms of which programming languages are designed. *Programming Language Implementation* should cover the design and implementation of compilers and interpreters. *Programming Language Specification* should cover syntax and semantics of programming languages. These three courses should demonstrate that programming languages are artifacts of interest in their own right. These courses should reinforce one another, and they should also reinforce the programming courses.

3 Programming 1 and 2

Programming courses typically have the following (very broad) learning objectives:

- to learn programming
- to learn a particular programming language.

Although CS is maturing as a discipline, I believe that it is still necessary to emphasize that learning how to *program* (in the sense of algorithm development and verification) should be the primary objective of a programming course; learning a particular *language* should be secondary. A student who

[2] Software engineering should *not* be equated with formal methods, which is only one aspect of it.

has learned and understood the essentials of programming can adapt relatively easily to new programming languages; a student who has learned only the superficial and idiosyncratic features of a particular programming language will find it difficult to adapt.

Nevertheless, CS educators must choose a particular language to be used as the medium to teach programming: a language in which to present examples and in which the students will attempt to solve programming problems themselves.

In a very different time and context, Ludwig Wittgenstein in *Tractatus Logico-Philosophicus* remarked:

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.

(The limits of my language define the limits of my world.) Wittgenstein's remark succinctly captures both the necessity of choosing a language and the importance of choosing the right one. Of course, Wittgenstein was thinking of natural languages and the language of mathematics, not programming languages, which then lay 40 years in the future. Nevertheless, his remark has been echoed by many CS educators, such as Alan Perlis and Niklaus Wirth.

Our collective experience does indeed suggest that the language in which a student first learns to program has a profound, perhaps indelible, influence on the way in which he or she thinks about programming. We have all heard tales, not entirely apocryphal, about students who, having learned BASIC at school, come to university and write BASIC-like programs in whatever language they are supposed to be using.

If we accept this argument, it follows that the choice of programming language(s) for *Programming 1* and *2* is critically important in designing a CS curriculum. There is no contradiction between this and my earlier contention that the primary objective is to learn how to program rather than to learn a particular language; like mathematical notation, the programming language is only a means to an end, but a very important means.

The issues of programming language choice, both in education and in the real world, generate a great deal of heat and (occasionally) some light. Indeed, a whole book [16] has been devoted to the issues (reflecting the state of the debate in the early 1990s).

With that and a few other honorable exceptions, the debate has too often been bedeviled by sales talk ('Language X is the solution to all your problems.') and holy wars ('Language Y is the only route to salvation!' 'Sacrilege! Y is the work of the devil!').

Here I will consider only conceptual, cognitive, pragmatic, resource, and transferable skills issues. These are the *real* issues in programming language choice.

- *Data:* tuples (or records); unions; arrays (or sequences); recursive structures.
- *Control:* composition (sequencing); selection; iteration; recursion; escapes.
- *Abstraction and parameterization:* procedures (or functions or methods or predicates); parameters; generics.
- *Encapsulation:* abstract data types and/or object classes.

Fig. 2. Suggested organizing concepts for *Programming 1* and *2*.

3.1 Conceptual issues

Given that the primary objective of *Programming 1* and *2* is to learn programming, educators should choose a programming language that supports and clearly exposes the fundamental concepts of programming: see Figure 2.

Data and control are central themes in all programming courses. The data and control structures listed in Figure 2 are uncontroversial. It is noteworthy that the first four control structures correspond one-for-one with the four data structures; for example, a tuple is a composition of data. This correspondence tends to confirm our intuition that these are the 'right' structures, not just an arbitrary wish-list.

Abstraction and parameterization provide the means to structure small programs. The simpler abstraction concepts – procedures and parameters – are central themes in all programming courses. Generics are a more advanced abstraction concept, and are supported by few programming languages, so it might be impractical to cover generics in an introductory programming course.

Encapsulation provides the means to structure large programs. There is now a consensus among CS educators that encapsulation is a concept of fundamental importance, not an advanced concept that can safely be neglected in introductory programming courses. Abstract data types or object classes (rather than individual procedures) are the proper building blocks for making large programs. Novice programmers cannot be expected to make their own building blocks until they have learned to write their own procedures, but they can and should be expected to use ready-made building blocks. For example, provided with a simple graphics class, novices can write simple programs with interesting (and directly visible) effects. Object-oriented programming languages tend to be particularly well equipped with such building blocks.

Students of *Programming 1* and *2* should not be expected to learn the concepts of Figure 2 in a theoretical sense. Instead, they should learn the manifestation of those concepts in the chosen programming language and (most importantly) how to deploy these concepts to solve practical programming problems.

For the purposes of teaching *Programming 1* and *2*, we should choose a programming language (or perhaps a pair of programming languages) that supports and clearly exposes all or nearly all of those concepts. That is to say, the programming language(s) should expose those concepts clearly, not conflating them or confusing them. A few examples should illustrate this point.

PASCAL has similar syntax for variable declarations and variable (reference) parameters, which tends to cause confusion. Both PASCAL and ADA conflate the distinct concepts of records and unions, making the latter concept in particular hard to learn. In other respects ADA supports the above concepts very well.

JAVA conflates all data structures as objects. Even simple records, with no operations other than accessing individual fields, demand a full-blown class declaration. Arrays are also objects in JAVA, although sufficiently distinctive to disguise this fact from novice programmers. Of course, different data structures are equipped with different methods, but only time will tell whether this conflation of data structures creates significant difficulties for novice programmers.

3.2 Cognitive (or paradigmatic) issues

I have already stressed that a student's first programming language has a profound influence on his or her understanding of programming. To an even greater extent this is true of the student's first programming *paradigm*: the conceptual distance between different paradigms is much greater than the conceptual distance between different languages in the same paradigm.

We can distinguish programming paradigms by the concepts that they emphasize:

- *Imperative* programming emphasizes procedures operating on (unencapsulated) variables.
- *Object-oriented* programming emphasizes methods operating on (encapsulated) objects.
- *Functional* programming emphasizes functions operating on immutable values.
- *Logic* programming emphasizes predicates operating on immutable values.
- *Concurrent* programming emphasizes processes exchanging messages.

Although a CS curriculum should expose students to all major paradigms, it would be a mistake to attempt too much too soon. *Programming 1* should introduce one paradigm and one language. The other paradigms should be gradually introduced in *Programming 2* and *Programming Language Concepts*.

It might be helpful to perceive a programming course as teaching a particular paradigm rather than a particular language. It should de-emphasize the particular features of the chosen language, covering them only to the extent necessary for students successfully to compile and run programs. It should emphasize the deeper concepts discussed in Section 3.1, which transcend the individual language. The programming course should make it relatively easy for students to learn other languages within the same paradigm.

A more specific cognitive issue is whether the chosen language is expressive and 'natural' for programming in its intended application area. For example, COBOL is claimed to be natural for business data processing, and FORTRAN for scientific computation; conversely, no-one would ever claim that either language was natural for writing systems software. This issue should be considered with care. Strong claims were made in the 1980s for the naturalness of pure functional programming languages. The notion of a program as a set of recursion equations is certainly appealing to CS academics, but many novice programmers find recursion unnatural, certainly less natural than iteration.

3.3 Pragmatic issues

Here I will consider a number of issues concerned with the practicalities of teaching and learning a particular programming language.

Error-proneness. The chosen programming language should not make it too easy for programmers to stray into error. A famous example of error-proneness was NASA's loss of a Venus probe in the 1970s, which turned out to be due to a single keystroke error (confusion of '.' with ',') in a large FORTRAN program. That was a syntactic idiosyncrasy, but some programming concepts are intrinsically error-prone. Goto-statements were found guilty as early as the 1960s. More recently it has become widely understood that explicit pointer manipulation is also error-prone, especially when combined with explicit deallocation.

Error-checking. The chosen programming language should provide comprehensive error-checking. In general, static checking is preferable to dynamic checking, which in turn is greatly preferable to no checking at all. A large proportion of all programming errors are type errors that can be detected statically, rather than being left to the vagaries of dynamic testing. Consequently, there is now a strong consensus among both CS educators and software engineers that programming languages should be statically-typed. We have seen many of the more popular dynamically-typed programming languages being overtaken by statically-typed successors, for example SMALLTALK by C++ and JAVA, and LISP by ML and HASKELL.

Subsetting. The chosen programming language should have clean subsets, to facilitate incremental learning. At any given time (even at the end of a programming course), a novice programmer has learned only a subset of the programming language. The programming language should therefore be designed in such a way that novices are unlikely to stray into unknown territory, with confusing or even disastrous consequences. PL/I was notorious for this problem. ADA (a language often unfairly compared with PL/I) is in fact well designed in this respect. For example, computing an out-of-range array index will cause an ADA program to halt and produce an error message, which is just what the novice would expect. The actual mechanism (the array-indexing operation raises an exception which, in the absence of a handler, propagates right out of the program) is transparent to the novice.

Support for good software design and implementation. Students will later encounter a thorough treatment of software design and implementation in *Software Engineering*. *Programming 1* and *2* should anticipate this by showing how to write maintainable (readable and modular) code. All modern programming languages make it possible to write readable code with well-chosen identifiers, literate comments, and good layout. However, many third-generation languages (such as PASCAL and C) do not adequately support modularity. *Programming 2* should show how to design modular programs whose building blocks are abstract data types or object classes, and how to construct such programs incrementally using separate compilation. The chosen programming language should therefore support these features.

Standardization. It is advantageous for students to learn a standard programming language. The existence of a language standard makes it more likely that different implementations are consistent with one another. In the 1960s and 1970s it was quite common for CS educators to use a locally-invented programming language for teaching. Naturally, they would claim that their local programming language was superior to the more widely-used programming languages. Whatever the merits of this claim, they were always outweighed by the drawbacks: the educators were committed to the costs of developing and maintaining an implementation of acceptable quality; and their students were very unlikely to encounter the local programming language later in their careers. Nowadays, such a claim would be completely unsustainable: there are plenty of good standard programming languages to choose from; implementation quality standards are much higher, and consequently development and maintenance costs are much higher too. Nonstandard language dialects should also be avoided. It is dispiriting for a novice programmer, having perhaps struggled to learn programming, later to be forced to adapt from a dialect to the standard language. A good example was TURBO PASCAL, which was surprisingly popular in the 1980s. This dialect adopted nonstandard semantics for input–output statements, sowing confusion in a part of the language that novice programmers already find hard to master.

3.4 Resource issues

The resources available for teaching are always constrained, and often inadequate. Resource issues inevitably play an important part in the choice of a programming language for teaching.

A sufficient pool of trained *staff* – classroom teachers and teaching assistants – who are familiar with the chosen programming language must be available. During a transition from one programming language to another, it is possible that a sufficient pool of trained staff will not yet be available, in which case a properly timed and resourced training program will be essential.

There should be at least one good-quality (and affordable) *textbook* to supplement the classroom teaching. The late-1990s growth of JAVA in education might have been even faster but for the initial lack of good-quality textbooks; fortunately, the situation is steadily improving.

There must also be at least one affordable high-quality *implementation* of the programming language. Nowadays, a stand-alone compiler is considered barely acceptable. Most educators and students prefer a program development environment, integrating an editor, compiler, interpreter, debugger, and (where appropriate) project management system. The implementation should be extremely robust: it will be more intensively used than any other compiler, and by students who will (accidentally) discover every possible way to break it! Ideally, the implementation should be portable, able to run on whatever variety of platforms happen to be available in the CS laboratories.

3.5 Transferable skills

The ability to program is a valuable transferable skill, i.e., a skill that will be useful in whatever career (academic or industrial, IT-centered or not) that the student chooses to pursue. Even for students who take only an introductory programming course, and who have not yet learned to adapt to different programming languages, the ability to program in a specific programming language is still potentially a useful transferable skill.

Some programming courses are designed for CS students only; others for non-CS students only; and others for a mix of students. It is vitally important that CS educators should consider the intended class of students when designing the course, and particularly when choosing the programming language to teach.

For *CS* students, the chosen programming language should be one that is suitable, not only for the immediate programming course, but also for more advanced CS courses and project work.

For *non-CS* students, the chosen programming language should be one that is immediately suitable for practical work in non-CS courses. For example, is the chosen programming language also available in the Physics and

Engineering laboratories? We cannot assume that non-CS students will easily adapt to a different programming language.

For *all* students, the chosen programming language should be one that is likely to be relevant in their future careers. This is a strong argument for choosing a standardized programming language.

3.6 Trends

We have now presented a large set of requirements for an introductory programming language. No programming language completely satisfies all these requirements; any choice is inevitably a compromise.

For one-and-a-half decades (*c.* 1975–90), PASCAL enjoyed a near-monopoly in programming courses throughout the world. This extraordinary phenomenon can be explained, in retrospect, by two main factors:

- Niklaus Wirth designed PASCAL specifically for educational purposes [15]. To a far greater extent than any contemporary programming language, PASCAL clearly exposed the fundamental concepts of data structuring, control structuring, abstraction and parameterization.
- A portable compiler kit made it easy to implement PASCAL on the many microprocessors that proliferated in the 1970s. Moreover, the resulting implementations were quite suitable for student use, up to contemporary standards.

By 1990, however, it was clear that PASCAL was well past its prime. CS educators increasingly understood the importance of encapsulation and incremental program development. In these and several other respects PASCAL was clearly deficient.

PASCAL's numerous offspring (such as MODULA-2, MODULA-3, OBERON, OBJECT PASCAL, DELPHI) attempted to remedy the lack of encapsulation, but none was sufficiently attractive to build a critical mass of support among CS educators (or any significant support in industry). ADA was seen by most educators as just too large and complex, and it lacked affordable implementations. C was even more deficient than PASCAL, and very error-prone.

C's offspring C++ is the subject of the most intense holy wars: its advocates point to its comprehensive support for object-oriented programming and its emerging industrial importance; its detractors underline its inelegant design and its error-proneness. The older object-oriented programming language SMALLTALK still had keen advocates, but Bertrand Meyer's new object-oriented programming language EIFFEL never attracted a critical mass of support.

Functional programming (represented by programming languages such as SCHEME, ML, and HASKELL) was strongly advocated by its supporters during the 1980s. Some introductory programming courses have adopted a functional programming language on the (rather negative) grounds that it is certainly different from any programming languages that the students have encountered at school. However, support for functional programming is always limited by its unnaturalness for programming applications with state.

For a long time it seemed unlikely that any programming language would ever inherit PASCAL's near-monopoly. A 1993 workshop at the UK's Open University [16] clearly exposed the lack of any consensus on the best introductory programming language, nor even on the criteria that should be used to choose such a programming language.

The situation changed dramatically in the later 1990s. The main driving force was the emergence of a clear consensus, in both industry and education, that object-oriented methods are the best way forward.

ADA is a serious contender now that several affordable and good-quality implementations are available. The language is very well designed, with excellent support for all the fundamental programming concepts, comprehensive error-checking, and so on. It is perfectly feasible to start by teaching a small (PASCAL-like) subset, and later introduce more advanced concepts such as exceptions, generics, and concurrency. ADA-95 has additional features that adequately support object-oriented programming. ADA-95 is a multi-paradigmatic programming language.

C++ is the most widely used object-oriented programming language in industry, and should be considered a serious contender in the context of a curriculum that emphasizes software engineering. Good-quality textbooks and implementations are available. Nevertheless, it remains error-prone, and some means must be found to protect novices from the worst traps. Experience with using C++ in an introductory programming course in a major university is reported in [7].

JAVA [4,6] is, of course, the major phenomenon of the 1990s. The language is compact, supports nearly all of the fundamental programming concepts, has comprehensive error-checking, and so on. Equally important, it is equipped with an incredibly rich class library.[3] The JAVA Virtual Machine concept makes *object code* portable everywhere, a unique achievement. The initial spartan implementation (JDK) has been joined by a variety of integrated program development environments. And the initial flood of low-quality hackers' guides has been superseded by a choice of good-quality textbooks [3,5,10,14]. JAVA has grown rapidly in popularity as an introductory programming language. Its growth rate is unprecedented: one leading textbook publisher estimates that JAVA is already taught in over half of all UK universities, and anecdotal evidence suggests that the situation is similar in other developed countries. It is not unreasonable to speculate that within a few years JAVA

[3] The JAVA class library is so rich, indeed, that teaching students how to use it effectively is itself an educational challenge.

will have achieved a near-monopoly in education.

4 Programming Language Concepts

The broad learning objectives for the *Programming Language Concepts* course should be:

- to deepen understanding of already-learned programming languages
- to acquire a conceptual framework for learning new programming languages
- to acquire a basis for understanding programming language implementations
- to acquire a basis for understanding programming language specifications.

The fundamental aim of this course is to present a conceptual framework for understanding programming languages, whether familiar or new. This aim is reflected the first and second objectives above, which are the key objectives. The third and fourth objectives relate this course to the courses to be discussed in Sections 5 and 6. Neither a programming language nor its implementation or specification can be properly understood in the absence of a conceptual framework.

In common use today there are perhaps twenty or thirty major programming languages, far too many to cover even superficially in a CS curriculum. A typical 'comparative programming languages' course therefore focuses on a small number of representative programming languages, including some that are new to the students. The expectation is that the experience gained by programming in the selected programming languages, as well as being valuable in its own right, will encourage students to compare and contrast these programming languages and thus understand the concepts underlying their design. In my view, this approach is flawed. A major programming language is a complex artifact that can be mastered only by time-consuming study and practice. Attention tends to be distracted by syntactic features and idiosyncrasies away from what really matters, the underlying semantic concepts. The sheer mass of detail hinders understanding of the conceptual relationships (both similarities and differences) between different programming languages.

Instead, *Programming Language Concepts* should focus on the concepts themselves. Wherever possible, programming languages already known to the students should be used to illustrate these concepts. Other programming languages should also be used for illustration where appropriate, but there should be no expectation that students learn these other programming languages in their entirety.

Figure 3 suggests a possible list of contents for *Programming Language Concepts*.

- *Data:* tuples (or records); unions; finite mappings (e.g., arrays); recursive structures.
- *Type systems:* static *vs* dynamic typing; monomorphic *vs* polymorphic typing; the type completeness principle.
- *Bindings:* bound and free identifiers; scope; static *vs* dynamic scoping.
- *Storage:* global, local, and heap variables; lifetimes; explicit *vs* automatic deallocation.
- *Control:* composition (sequencing); selection; iteration; recursion; escapes; concurrency.
- *Abstraction:* procedures (abstractions over commands); functions (abstractions over expressions); generics (abstractions over declarations); the abstraction principle.
- *Parameterization:* parameter mechanisms; the correspondence principle.
- *Encapsulation:* information hiding; abstract data types; object classes and subclasses; single *vs* multiple inheritance.
- *Paradigms:* imperative programming; object-oriented programming; concurrent programming; functional programming; logic programming.

Fig. 3. Suggested contents of *Programming Language Concepts*.

It should come as no surprise that the suggested organizing concepts for *Programming 1* and *2* – data, control, abstraction and parameterization, and encapsulation (see Figure 2) – also feature prominently in *Programming Language Concepts*. In *Programming 1* and *2*, these concepts should be implicit, serving only as a framework for designing the courses; in *Programming Language Concepts* they should be made explicit. The remaining concepts listed in Figure 3 – type systems, bindings, storage – should also be made explicit in *Programming Language Concepts*, but are not useful as organizing concepts for a programming course since they are best explained by comparing and contrasting two or more different programming languages.

Most of the concepts listed in Figure 3 are very straightforward, and students should be able to recognize them in familiar programming languages. Much more difficult, however, is to understand what distinguishes well-designed programming languages from badly-designed ones. Landin, Tennent, and others have proposed several semantic principles [11,12], such as the abstraction principle, the correspondence principle, and the type completeness principle, which throw light on this issue. Understanding these principles and their implications is a challenge for even the best students.

The CS curriculum should include experience of programming in all the major programming paradigms. As explained in Section 3.2, each paradigm is characterized by the concepts that it chooses to emphasize. *Programming Language Concepts* is a good place explicitly to compare and contrast the major paradigms, and this is an opportunity for practical work in what would otherwise be a theoretical course. For example, students who have practised only imperative and object-oriented programming should be given the opportunity to write simple functional and logic programs. (Note that this does *not* entail learning a functional or logic programming language in its entirety.)

The suggested contents of *Programming Language Concepts* are covered well by several textbooks, for example [9,11,12].

By its very nature, *Programming Language Concepts* is concerned largely with unchanging principles. What does change over time is the relative importance of different concepts and paradigms. *Programming Language Concepts* should reflect the current widespread interest in object-oriented programming by proper attention to this paradigm and its underlying concepts.

5 Programming Language Implementation

The learning objectives for the *Programming Language Implementation* course should be:

- to deepen understanding of programming language pragmatics
- to understand run-time organization for different programming languages
- to understand the functions of compilers and interpreters, and their interactions
- to understand the internal structures of compilers
- to understand basic compiler, interpreter, and run-time algorithms.

All programmers should have some understanding of how programming languages are implemented. The internal structures of a compiler (i.e., its top-level decomposition into syntactic analysis, contextual analysis, and code generation) underlies compile-time checking and diagnostics. Run-time organization (i.e., data representation, the run-time stack, the heap, garbage collection, etc.) similarly explains the nature of run-time checking and diagnostics.

A good understanding of run-time organization makes us better programmers, able to select different programming constructs in full knowledge of their costs. The pragmatic aspects of programming languages are just as important as their syntax and semantics. A good example is garbage collection, which is needed to implement many programming languages (including JAVA but not C++), and which is becoming increasingly important in view of the emerging

- *Compilers and interpreter interactions:* functions of compilers and interpreters; multi-stage compilers; interpretive compilers; portable compiler kits; bootstrapping.
- *Run-time organization:* data representation; the stack, frames, dynamic and static links; the heap and garbage collection; run-time organization for object-oriented programming languages.
- *Compiler structure:* syntactic analysis, contextual analysis, code generation, and code optimization phases; single-pass *vs* multi-pass compilation.
- *Syntactic analysis:* scanning; parsing.
- *Contextual analysis:* identification; type-checking.
- *Code generation:* storage allocation; stack-machine code generation; register-machine code generation, register allocation and assignment.
- *Code optimization:* peephole optimization; folding; constant propagation; common subexpression elimination; code motion.

Fig. 4. Suggested contents of *Programming Language Implementation*.

consensus among programmers and language designers that the benefits of automatic storage recycling outweigh its costs.

Programming Language Implementation should demonstrate how compilers and interpreters can be deployed to make multi-stage compilers, to make portable compiler kits, and to bootstrap compilers. The advent of JAVA and the JAVA Virtual Machine (JVM) in the 1990s has revived interest in abstract machines, interpreters, and portable compilation. The JAVA phenomenon was in many ways reminiscent of the Pascal phenomenon in the 1970s, which was largely due to the Pascal/P-code portable compiler kit.

Programming Language Implementation should also introduce students to some basic algorithms used in programming language implementation. Parsing is important in many applications as well as compilers, and the recursive-descent parsing method is easily understood. Similarly, interpreting algorithms crop up in many applications.

Figure 4 suggests a possible list of contents for *Programming Language Implementation*. Unfortunately, textbooks suitable for an introductory course [13] are few and far between; however, some programming language concepts books [9] also cover run-time organization.

Programming Language Implementation is deliberately suggested as an introductory course. It should explain the functions of the principal compiler phases – syntactic analysis, contextual analysis, code generation and optimization – but it need not cover all these topics in detail. Few programmers will ever write a compiler, so a sophisticated understanding of compiling algorithms, although instructive, is not essential. In particular, the course should explain the basic ideas of code generation and optimization, but the actual algorithms are both difficult and target-dependent and are probably better omitted. Instead, *Programming Language Implementation* could be followed by a more specialized *Compilers* course, perhaps offered as an elective for those students who seek an in-depth treatment of compiler technology. Suitable textbooks for a more specialized course include the classic [1] and the more recent [2].

6 Programming Language Specification

The broad learning objectives for the *Programming Language Specification* course should be:

- to understand the relative merits of informal and formal specification
- to deepen understanding of programming language concepts
- to acquire a basis for predicting program behavior without recourse to a particular implementation
- to acquire a basis for reasoning about programs.

All CS students should understand the difference between informal and formal specifications, and the advantages and disadvantages of each. The specification of software should be covered in the context of *Software Engineering*. The specification of programming languages is a more specialized topic, but all CS students should at least be able to read and understand a semiformal language specification such as [6]. In particular, they should be able to predict the behavior of a given program without having to test it on a particular implementation. They should also understand that the same program might behave inconsistently on different platforms because of platform dependencies in the program itself, or because of implementation differences due to ambiguities in the language specification (as well as because of implementation faults).

All CS students should understand the distinction between syntax, static semantics, and dynamic semantics. They should also understand their relationship with language implementation: a compiler's syntactic analysis phase enforces the source language's syntax; its contextual analysis phase enforces the language's static semantics; its code generation and optimization phases respect the language's dynamic semantics.

In nearly all published language specifications, only the syntax is specified formally. Every CS student should be familiar with regular and context-free grammars and their notations.

- *Formal vs informal specification.*
- *Syntax:* BNF used for phrase-level syntax; regular expressions used for lexical-level syntax; EBNF.
- *Static semantics:* natural semantics used for typing rules.
- *Dynamic semantics:* natural semantics used for evaluation rules.

Fig. 5. Suggested contents of *Programming Language Specification.*

Formal specification of semantics is often omitted altogether in the CS curriculum. Nevertheless, I would recommend at least an introduction to natural semantics. This formalism is, as its name suggests, very natural for expressing both static semantics (typing rules) and dynamic semantics (evaluation rules). Natural semantics is easily understood by students familiar with predicate logic and/or logic programming. Furthermore, natural semantics can easily be prototyped in a programming language like Prolog.

Figure 5 suggests a possible list of contents for *Programming Language Specification.*

A more specialized course should include a deeper study of natural semantics, and a comparative study of another formalism such as structural operational semantics, denotational semantics, or even action semantics. A suitable textbook for such a course would be [8].

7 Conclusions

CS education has developed to the stage at which we can design an integrated sequence of courses about programming and programming languages, based on a set of programming concepts that is both stable and well understood. The concepts themselves are unchanging, but the relative importance of different concepts does vary over time.

The most significant trend since the 1990s has been an almost universal convergence on object-oriented methods as the software technology of choice, in both industrial and educational sectors. This has been accompanied since the late 1990s by an inexorable advance of JAVA towards a near-monopoly position as the language of choice for teaching programming.

There is no sign that these trends will weaken in the foreseeable future. On the contrary, they will be strengthened by generations of CS graduates moving into industry, already skilled in object-oriented methods in general and possessing a deep understanding of object-oriented programming in particular.

Acknowledgements

I wish to thank Deryck Brown, and participants at SBLP'00, for their helpful comments and feedback on the talk that formed the basis of this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman: *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts (1986).

[2] A.W. Appel: *Modern Compiler Implementation in Java*, Cambridge University Press, Cambridge, England (1998).

[3] D.A. Bailey and D.W. Bailey: *Java Elements – Principles of Programming in Java*, McGraw-Hill, Singapore (2000).

[4] D. Flanagan: *Java in a Nutshell*, 3rd edition, O'Reilly & Associates, Sebastopol, California (1999).

[5] M.T. Goodrich and R. Tomassia: *Data Structures and Algorithms in Java*, Wiley, New York (1998).

[6] J. Gosling, W. Joy, and G. Steele: *The Java Language Specification*, 2nd edition, Addison-Wesley, Reading, Massachusetts (1996).

[7] P.A. Lee and R.J. Stroud: C++ as an introductory programming language, in [16], pages 63–82.

[8] H.R. Nielsen and F. Nielsen: *Semantics with Applications*, Wiley, Chichester, England (1992).

[9] R. Sethi: *Programming Languages – Concepts and Constructs*, Addison-Wesley, Reading, Massachusetts (1989).

[10] T.A. Standish: *Data Structures in Java*, Addison-Wesley, Reading, Massachusetts (1998).

[11] R.D. Tennent: *Principles of Programming Languages*, Prentice Hall International, Hemel Hempstead, England (1981).

[12] D.A. Watt: *Programming Language Concepts and Paradigms*, Prentice Hall International, Hemel Hempstead, England (1990).

[13] D.A. Watt and D.F. Brown: *Programming Language Processors in Java – Compilers and Interpreters*, Prentice Hall International, Harlow, England (2000).

[14] D.A. Watt and D.F. Brown: *Java Collections – an Introduction to Abstract Data Types, Data Structures, and Algorithms*, Wiley, Chichester, England (2000).

[15] N. Wirth: The programming language Pascal, *Acta Informatica* 1, pages 35–63 (1971).

[16] M. Woodman (editor): *Programming Language Choice*, International Thomson Computer Press, London, England (1996).